Protocol Composition Frameworks
A Header-Driven Model*†

Daniel C. Bünzli, Sergio Mena, Uwe Nestmann
École Polytechnique Fédérale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland
daniel.buenzli, sergio.mena, uwe.nestmann @epfl.ch

Abstract

Protocol composition frameworks provide off-the-shelf composable protocols to simplify the development of custom protocol stacks. All recent protocol frameworks use a general-purpose event-driven model to manage the interactions between protocols. In complex compositions, where protocols offer their service to more than one other protocol, the one-to-many interaction scheme of the event-driven model introduces composition problems by mixing up the targets to which data (list of headers) should be delivered. To solve these problems, we propose to shift the driving force behind interactions from the events to the headers they carry. We show that the resulting domain-specific header-driven model solves the composition problems, provides statically typed header handling and enhances protocol readability.

1 Introduction and summary

A protocol stack is a middleware infrastructure that provides a service to simplify the development of distributed applications running on a failure-prone computing infrastructure (communication or node failures). Whereas a protocol stack simplifies the development of a distributed application, the purpose of a *protocol composition framework* [5, 9, 4, 8, 1] is to ease the development of custom protocol stacks.

The vision of being able to tailor stacks to the needs of specific applications by composing off-the-shelf protocols drives research in protocol frameworks. This vision will only be realized if protocols are developed modular and hierarchical wise and powerful component languages are provided to structure and compose them (§2, [2, §7.1]). In this paper we argue that these goals are hindered by the programming model of recent protocol frameworks (§3). These frameworks [8, 4, 1] all base the interaction mechanism between protocols on an *event(-driven) model*. In this model, computations are specified by event handlers. Handlers can be bound to events and are executed when the latter are triggered. *Many* handlers can be bound to a *single* event, which means that the interaction scheme is *one-to-many* (vs. *one-to-one*).

The structure of protocol compositions is changing from *stacks* to *graphs* [6, 7] in which protocols offer their services to *more than one* protocol. The event model matches the reactive nature of distributed computing and the way protocols are described in the literature, but its one-to-many interaction scheme introduces composition problems in protocol graphs. Protocols may receive events that are not targeted at them. This compromises the definition of powerful component languages on top of an event model because *ad hoc* mechanisms need to be introduced to "route" events to the right protocols. Moreover, the event model doesn't properly handle *peer interactions*, where a protocol interacts with its peer running on another node by using the service of a lower-level protocol. The way events handle this ubiquitous pattern is (1) complex, many unnecessary bindings need to be done by the *composer* (2) obscure, the indirections introduced by events hide, in the code, the logical structure of peer interactions (3) unsafe, misbindings may lead to runtime type errors or erratic behaviour.

Instead, we propose a novel and simple alternative (§4, [2, §6]) that shifts the driving force behind interactions from events to the headers they carry. The resulting *header-driven model* (1) solves the composition problems of the event model (2) simplifies inter-protocol dependencies (3) concisely handles peer interactions and explicitly reveals their logical structure (no obfuscating indirections) (4) provides better static typing which avoids the runtime type errors and erratic behaviour that can occur in the event model. We show how the two models compare in the context of a component language in [2, §7].

*Supported by the Swiss National Science Foundation, grant No. 21-67715.02 and the Hasler Foundation, grant No. DICS 1825.

†Due to an unexpected cut in the number of pages, the paper omits many details. For a better understanding of the material we strongly encourage the reader to consult the technical report [2] (10 pages).

The contributions of this paper are (1) the demonstration that the event model[1] is *not* the right programming model for protocol composition frameworks because of the mix-up of events in protocol graphs and the unsatisfactory handling of peer interactions (2) the proposal of an alternative, header-driven model solving these problems and with better compositional properties.

Acknowledgments. We thank Christophe Gensoul for implementing the first NUNTIUS prototype and Rachele Fuzzati, Olivier Rütti, André Schiper, Paweł Wojciechowski for their feedback.

2 Requirements

Composition. Previous research [6, 7] shows that the modular decomposition of group communication middleware results in complex interactions between various components (protocols) from different abstraction levels. These compositions are out of the scope of simple stack-based composition schemes. In particular, services of protocols are sometimes used by *many* other components, leading to graph-based composition schemes.

In order to tackle this complexity, a powerful component language is needed. This language should allow the definition of parametric and hierarchical components and provide information hiding mechanisms to hide complex compositions. Composing protocols should be an easy task. In particular we would like to have coarse-grained composition mediated by interfaces instead of fine-grained sequences of input-output binding instructions.

Hence we are looking for a *programming model* that supports, on top of it, the definition of a component language satisfying these requirements.

Peer interaction. Let us respectively refer to a protocol requesting a service of another as the *client* and the *server* protocol. A particular and recurrent interaction seen between protocols is *peer interaction*. Conceptually, in a peer interaction, a protocol A interacts with its peer as if it was performing a remote procedure call (Fig. 1, left). Concretely, A issues a request to a local server protocol C, which results in an interaction between their peers (Fig. 1, right). An example is the send/deliver scheme of a communication protocol.

Most protocols work by interacting with their peers. These interactions almost always occur via peer interactions to benefit from the properties provided by other protocols. As such, expressing this pattern in our programming model should be clear, explicit and concise.

[1] Which should not be equated with a *reactive* programming model, our proposal is also reactive.

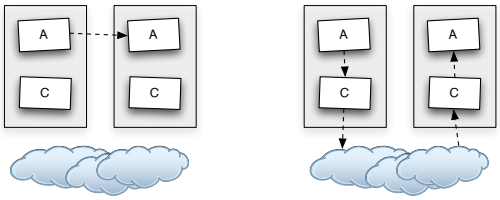


Figure 1. Peer interaction, execution paths

3 Misfit of the event model

Event-driven primitives. In an event-based programming model a program is a sequence of event definitions, event handler definitions and event bindings. The latter bind handlers to specific events. When an event is triggered, all handlers bound to it are executed thereby possibly triggering new events. In this model, components aggregate and structure events and event handler definitions as well as event bindings.

The discriminating feature of the event model is that *none* or *many* handlers can be bound to a *single* event and vice versa. Many semantical variations can be devised on these constructs. Static versus dynamic binding, serial versus concurrent execution of handlers, etc, but these differences are not relevant to our discussion. Note that in practice, protocol frameworks [4, 8] may use additional interaction schemes and structuring entities (e.g. channels [8]).

On the one hand, in stack-based compositions of protocols one does not use the one-to-many interactions provided by the event model. On the other hand, this interaction scheme is not adapted to graph-based compositions where a server protocol offers its service to more than one client protocol.

The event routing problem. The problem lies in the way a server protocol responds to a request issued by a client protocol. The server's response is returned by triggering an event to which the client can bind a handler. Now if two protocols A and B use the same service provided by C (Fig. 2, left), they will both bind an event handler to this event. If A issues a request on C, both A and B will get the result (Fig. 2, right). The problem is that this behaviour is in *most cases* unwanted. It seems quite natural that requests issued by two unrelated clients should not interfere. Thus in the example above a mechanism needs to be introduced so that B can discard the (wrong) data it gets.

Of course the event routing problem does not show up in stack-based compositions since each protocol has at most one client. But in complex graph-based compositions [7], this problem occurs frequently. In a component language with parametric components, the problem is even more

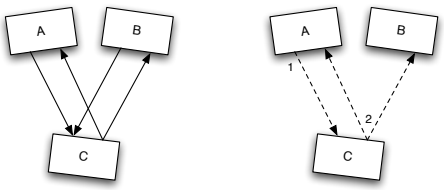
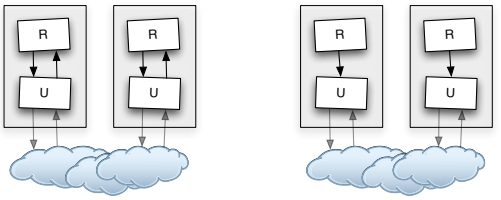


Figure 2. The event routing problem



Event model Header-driven model

Figure 3. Composition bindings

acute: a protocol C can easily be given as a parameter to more than one parametric protocol and the usage of C'service by those should also not interfere.

***Ad hoc* solutions.** There are solutions to this problem in the event model. However, these solutions are *ad hoc* and neither satisfactory nor elegant (see [2, §4]).

Wrong interaction scheme. The event model allows more complex interaction patterns than, for example, function application. But it does not match what we need most of the time. It has been argued [6] that most bindings are *one-to-one*, a single handler is bound to a single event. Moreover, the indirections introduced by the event binding mechanism significantly complicate and obfuscate the implementation and composition of protocols (see §4, [2, §7]). Finally, if really needed, a one-to-many pattern is easy to implement on top of a functional or object-oriented language.

The event model can be seen as an *observer* pattern [3]. The intent of this pattern is to "define a one-to-many dependency between objects such that when one object changes state, all its dependents are notified and updated automatically". The use of this pattern in our setting is clearly not appropriate since, as we said above, state changes of a server protocol should *not*, in most cases, affect all its clients.

The observer pattern is also used to develop loosely coupled components. But the modular decomposition of a complex protocol results in *tightly* coupled components, in the sense that the role and properties of the sub-protocols are clearly defined.

4 The header-driven model

The compositional shortcomings of the event model lead us to seek a new programming model for protocol frameworks. By contemplating how badly the event model manages peer interactions, we get to our new proposal.

Peer interaction in the event model. Protocols typically encapsulate communication data for their peers in messages and headers. A *message* is a list of headers and a *header* is a typed container for data. In a peer interaction a sequence of protocols is traversed. Starting in the first protocol with the empty message, each protocol pushes a specific header onto the message with the data for its peer and triggers an event to pass the resulting message to the next protocol. The last protocol of the sequence sends the message to the peer node with inter-node communication. On the peer node, the reversed sequence of protocols is traversed (provided bindings are correctly specified). Each protocol pops from the message the header transmitted by its peer and triggers an event to pass the resulting message to the next protocol.

A concrete example is given by the composition depicted in Fig. 3, left. Black arrows denote the explicit bindings — dependencies — specified to compose the two protocols. Protocol R offers reliable node-to-node communication and U unreliable node-to-node communication. Conceptually R wants to give data to its peer by "calling" a handler `recv` on the other node. Protocol R packs data onto a message and gives it to the lower level component U by triggering an event. U sends the message with inter-node communication. Upon reception on the peer node, U's peer triggers an event, *hopefully* bound at composition time to R's `recv`, to deliver the message to R. The way the event model manages peer interactions has the following drawbacks.

- *Compositionally suboptimal.* The designer of R knows that `recv` should handle the data given to U. But, because of the indirections introduced by the binding mechanism, this cannot be explicitly coded in the protocol. Instead, the designer writes the handler `recv` and *hopes* that the right bindings will be done during composition. This is compositionally suboptimal because a constraint known at design time has to be explicitly enforced later, at composition time.

  Furthermore, handler `recv` is completely internal to the protocol R (and its peers). Yet it needs to show up in the *interface* of the protocol so that it can be bound by the composer to an event of U. This goes against abstraction since it prevents information hiding.

- *Event routing problem.* If U (or R) is used by more than one client we get routing problems.

- *Failing or mixed up header deconstruction.* Messages are heterogeneous lists of data. They cannot be given

a more informative type than "msg". A composer may incorrectly bind recv to an event unrelated to U but whose type matches msg. In that case, two bad things may happen. Either a *runtime type error* occurs because recv tries to pop a header from a message whose structure does not match its expectations. Or recv pops a header of the right type but that is not intended for R possibly resulting in *erratic behaviour*.

From events to headers. Protocols often use a single handler to manage the reception of peer interactions. Nevertheless, some protocols need to send different kind of data via this single handler. In order to do so, they introduce a *tag name* in the header to indicate the kind of information they transmit. Since a header usually remains internal to a protocol and its peers, it is not restrictive to impose that each header shall be *named*, and that each name shall be *declared by at most one protocol* in a composition. A composition satisfying these constraints has the following interesting property. If we look at the names of a message's sequence of headers, we can approximately see the sequence of protocols — the route — that will handle the message when it is processed by the peer composition. This means that there is no need, for the composer, to explicitly bind the upward flow of events. In other words *the message's sequence of headers drives its processing in the protocol graph*.

The event model prevents us from exploiting this property. Thus, instead of having events at the core of our interaction scheme, we should have *headers*. This is the essence of our proposal.

The essential ingredients of a *header-driven model* are headers and messages. A message is a list of headers. Headers are *named* containers carrying statically typed data. To construct a header, its name must be defined. A header handler defines a header name and associates a computation to the deconstruction of all messages starting with that name. Message dispatch is the interaction scheme, it deconstructs messages. When a message is dispatched, the unique header handler corresponding to the head of the message is invoked with the head's data and the tail of the message as arguments. Compared to the event model we can say that (1) header handlers replace event handlers (2) message dispatch replaces event triggering and (3) the event binding mechanism is dropped.

In a header-driven model, the peer interaction described above occurs as follows. Protocol R pushes data onto a message using a header recv. Unlike in the event model, this *specifies* which header handler should be invoked on that data at the peer node. R gives this message to U using, for example, function application. U sends the message with inter-node communication. Upon reception on the peer node, U dispatches the message which becomes automatically deconstructed at the right place by invoking the unique header handler recv.

Mirroring the defects of the event model, our proposal has the following benefits.

- *Better compositional properties.* Messages "know" where they need to be deconstructed. Therefore no bindings for the upward control flow need to be specified. This removes one dependency between the two components (Fig. 3, right). Besides, handler recv becomes truly internal to the protocol R and its peer, it does not appear in the interface.
- *No event routing problem.* If another protocol S uses U, it packs data into one of its own headers h. When U's peer dispatches that message, it is automatically routed to the handler h of the peer of S.
- *Correct header deconstruction.* A header is always constructed with the right type in the scope of a header handler for it. For any header occurring in any message, there is exactly one corresponding handler. This implies that at runtime, neither the deconstruction of a message can fail, nor can a handler get unrelated data. Both the runtime type errors and erratic behaviours found in the event model cannot occur.

For concrete informations about header-driven primitives and their implementation see [2, §6].

References

[1] F. Brasileiro, F. Greve, F. Tronel, M. Hurfin, and J.-P. Le Narzul. Eva, an event-based framework for developing specialized communication protocols. In *Proc. IEEE NCA'01*, 2001.

[2] D. C. Bünzli, S. Mena, and U. Nestmann. Protocol composition frameworks, a header-driven model. Technical Report IC/2005/007, http://ic.epfl.ch, Epfl, 2005.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[4] M. A. Hiltunen and R. D. Schlichting. A configurable membership service. *IEEE Transactions on Computers*, 47(5):573–586, 1998.

[5] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[6] S. Mena, X. Cuvellier, C. Grégoire, and A. Schiper. Appia vs. cactus : Comparing protocol composition frameworks. In *Proc. of SRDS*, 2003.

[7] S. Mena, A. Schiper, and P. T. Wojciechowski. A step towards a new generation of group communication systems. In *Middleware 2003*, volume 2672 of *LNCS*, 2003.

[8] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. of ICDCS'01*, pages 707–710, 2001.

[9] J. Pereira and R. Oliveira. Object-oriented open implementation of reliable communication protocols. *OOPSLA'97 Ws. on Dependable Distributed Object Systems*, Oct. 1997.